// Mesh size; 129 vertices per di
for (int d = 0; d < Dim; d++) ver
// Uniform, logically-rectilinear

POOMA
Parallel Object-Oriented Methods and Applications

What is POOMA?

POOMA (Parallel Object-Oriented Methods and Applications) is an object-oriented framework for applications in computational science requiring high-performance parallel computers. It includes C++ classes designed to represent common abstractions in these applications—abstractions such as arrays, fields, geometries, meshes, and particles.

The main goals of the POOMA framework include the following:

- Code portability across serial, distributed, and parallel architectures with no change to source code
- Development of reusable, cross-problem-domain components to enable rapid application development
- Code efficiency for kernels and components relevant to scientific simulation
- Framework design and development that can be driven by applications from a diverse set of scientific problem domains
- Shorter time from problem inception to working simulations

The earlier version of POOMA, generally referred to as POOMA r1, has enjoyed considerable success in meeting these goals for a variety of applications, including multi-material compressible hydrodynamics, accelerator modeling, and Monte Carlo transport.

POOMA 2.x is the next generation of POOMA software, designed to take advantage of advances in C++ compiler technology and multi-threaded operation. As a result, it is more flexible, extensible, and efficient than POOMA r1.

The Array Abstraction

The POOMA multidimensional *Array* concept is loosely based on the FORTRAN 90 built-in array facility, but differs in a few significant ways. First, FORTRAN arrays support domains consisting of the tensor product of N one-dimensional discrete index sequences, each having unit stride, and ranges described by various sizes of integers, floating point numbers, and complex numbers. POOMA extends the array concept to additionally support bounded, continuous (floating point) N-dimensional domains and arbitrary ranges described by user-defined types. Second, unlike FORTRAN, array indexing in POOMA is polymorphic; that is, the indexing operation X(i1,i2) can perform the mapping from domain to range in a variety of ways, depending on the particular type of array that is being indexed.

Built-in FORTRAN arrays are dense and the elements are arranged according to column-major conventions. However, as Figure 1 shows, FORTRAN-style "Brick" storage is not the only format of interest to scientific programmers. For compatibility with C, one might use an array featuring row-major storage. To save memory or operations, it is advantageous to use an array with a data structure that can automatically compress itself to a single value if all the element values are the same. To exploit parallelism, it is convenient for an array's storage to be broken up into patches, which can be processed independently by different CPUs in a shared-memory multiprocessor. In a cluster environment, these patches can reside in different memory spaces. Finally, one can imagine an array with no data at all. For example, the values can be computed from an expression involving other arrays or computed analytically from the indices.

POOMA 2.x is the next generation of POOMA software, designed to take advantage of advances in C++ compiler technology and multi-threaded operation.

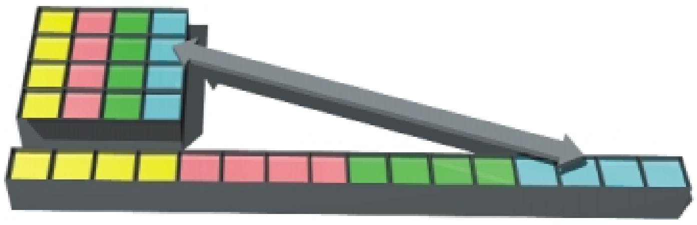
The POOMA *Array* class is templated for flexibility and extensibility:

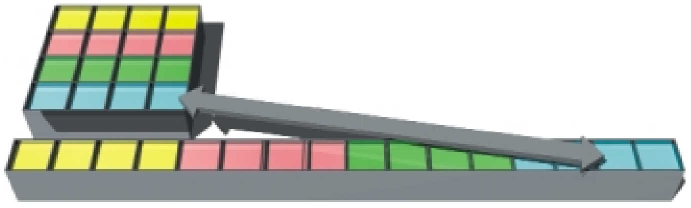
```
template <int Dim, class T = double, class EngineTag = Brick>
class Array;
```

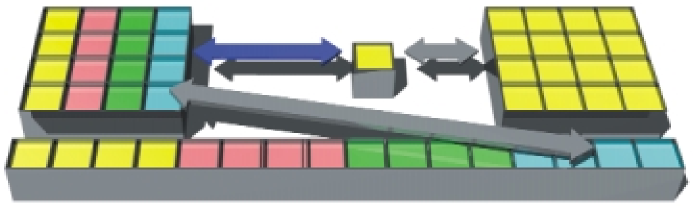
The template parameters *Dim*, *T*, and *EngineTag* determine the precise type of the *Array*. *Dim* represents the dimension of the *Array's* domain. *T* gives the type of *Array* elements, thereby defining the output range of the *Array*. *EngineTag* specifies the types of the indices and the manner of indexing. These template parameters allow for arbitrary domain, arbitrary range, and polymorphic indexing.

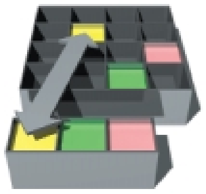
In addition to the intrinsic types like *double* and *int*, POOMA supplies fixed-size *Vector*, *Tensor*, and *Matrix* classes for use as the *T* parameter of *Arrays*. POOMA also supplies over 20 "Engines" capable of doing indexing in different ways. These can be plugged into an *Array* by simply changing the *EngineTag* template parameter.

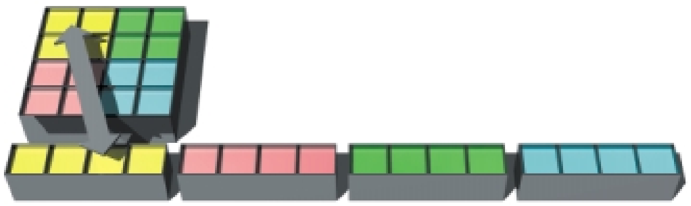
When used by itself, an *Array* object *A* refers to all of the values in its domain. Elementwise mathematical operations or functions can be applied to an *Array* using straight-forward notation, like *A* + *B* or *sin(A)*.


Brick: FORTRAN-style column-major storage.


C-style Brick: row-major storage.


Compressible Brick: a Brick that can compress to a single value to save storage and operations.


Sparse: store non-zero elements only.


Multipatch: parts reside in different memory spaces.

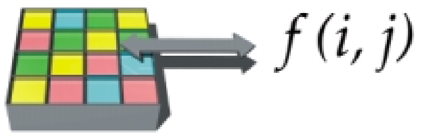


Figure 1. Some indexing examples of interest to scientific programmers.

Expressions involving *Array* objects are themselves *Arrays*. The operation *A(d)*, where *d* is a *Domain* object that describes a subset of *A*'s domain, creates a view of *A* that refers to that subset of points. Like an *Array* expression, a view is also an *Array*. Changes to a view of *A* modify the original *Array A*. If the domain *d* represents a single point in the domain, this indexing operation returns a single value from the range. Equivalently, it is possible to index an N-dimensional array by specifying N indices. Reduction operations, which work in parallel, are supported as well as "stencil objects" that are used to create high-performance finite-difference operators.

The Field Abstraction

The POOMA *Field* abstraction maps points in a coordinate-space domain to values. It contains a representation of the spatial domain and can specify what happens at the boundaries of that domain. The *Field* class template is written

```
template<class Geometry, class T, class EngineTag>
class Field;
```

The *T* and *EngineTag* parameters are the same as in *Array*. The *Geometry* parameter is new and it represents the coordinate-space domain. POOMA currently provides classes that represent only discrete geometries—specifically, sets of points defined on a variety of centering positions relative to logically rectilinear meshes. POOMA also currently supports cartesian and cylindrical coordinates in multiple dimensions.

This design of *Field* is very general. It admits continuous as well as discrete domains given appropriately defined classes for the *Geometry* and *EngineTag* parameters. By making no assumptions about underlying meshes or discretization schemes, it allows the possibility of implementing almost any kind of field-based numerical scheme.

POOMA has a general scheme for defining boundary conditions and automatically applying them when needed. A subset of a *Field's* domain is defined as a "boundary" whose values are to be computed based on the values of some other observed subset. Whenever the boundary values are read, POOMA checks whether the observed values have changed since the last boundary condition application. If so, it applies the boundary conditions again. The POOMA boundary condition implementation supports boundary conditions that do not require an observed set along with non-automatic boundary conditions embedded in user-defined operators. It is also possible to define internal boundary conditions.

POOMA currently supplies several canned external boundary conditions, implemented using guard (ghost) layers of *Field* elements for discrete *Fields*. These include typical reflecting, constant, periodic, and linearextrapolation boundary conditions. It is straightforward for the user to add boundary conditions using a high-level data-parallel application programming interface, including spatially-dependent and time-dependent boundary conditions.

Like *Arrays*, POOMA *Fields* support elementwise operations and functions, views, indexing, reduction operations, and some high-level finite-difference-based differential operators for divergence, gradient, and averaging. The example code shown in Figure 2 illustrates some of the specific *Geometry* and other *Field*-related classes and mechanisms provided by POOMA.

The Particles Abstraction

Particle simulation techniques are used in many different sorts of scientific applications, including Monte Carlo sampling, particle-in-cell simulation, and molecular dynamics. POOMA supports the wide variety of particle simulations with a set of classes that provide storage for particle data, strategies for distributing that data, and the ability for particle data to be filtered or to interact with field data.

The *Particles* class in POOMA is a container of arrays of particle data, where each array stores the values of one particle characteristic for all particles. These arrays are represented by the *DynamicArray* class:

```
template <class T, class EngineTag>
class DynamicArray : public Array<1,T,EngineTag>
```

A *DynamicArray* is a 1D array that can create or destroy elements on the fly. Each *DynamicArray* in a *Particles* object can represent a particle attribute using a different type *T*. Besides storing the particles attributes, *Particles* also provides the interface for creating and destroying particles, enforcing a data layout strategy, and applying boundary conditions, or "filters," to the particle attributes. POOMA has a variety of boundary condition types, such as periodic, reflecting, and absorbing, that modify a particle when its value for some attribute exceeds a prescribed range.

The *Particles* class is declared as

```
template <class PTraits> class Particles;
```

where *PTraits* is a "traits" class defining certain characteristics of particles. By convention, the *PTraits* template argument must provide C++ typedefs for the *EngineTag* of all the *DynamicArrays* contained by the *Particles* class and for the type of layout strategy to be employed.

POOMA provides several canned *PTraits*-type classes that provide these typedefs, but the user is free to create new classes that may contain additional traits information.

The layout strategy controls decomposition of the particle *DynamicArrays* into separate patches. POOMA currently offers two particle layout strategies: *UniformLayout*, which divides the particle data into evenly sized patches, and *SpatialLayout*, which decomposes the data such that particles with positions inside a given patch of a POOMA *Field* are grouped together. This second strategy is most useful for cases having particle-particle or particle-field interactions.

The intent of the *Particles* class is to provide the user with a simple interface to their own customized *Particles* class. The user derives a new class from *Particles* that contains the desired particle attributes as member data. The user's subclass will inherit the public interface of the *Particles* class, which is needed in order for POOMA to manipulate the attributes. For example, the user can define a class called *Airplanes* that contains attributes for each airplane's current coordinates, velocity, and remaining fuel as shown in Figure 3.

To simplify interpolation between particle positions and *Field* elements, POOMA provides global gather/scatter functions and a variety of interpolation stencils. Each gather or scatter call takes a particle attribute (or possibly a scalar value), a *Field*, and a position attribute, and automatically performs the interpolation in a data-parallel fashion.

For instance, if we wanted to scatter the number density of our airplanes into a *Field* called "pdens," we would say

```
scatterValue(1,pdens,p.coordinates(),NGP());
```

Here we scatter a value of "1" for each airplane into the *Field* element of "pdens" nearest to each airplane's coordinates. We are using the nearest-grid-point interpolation scheme, as indicated by the temporary object of type NGP that is passed to the scattering function.

POOMA 2.2 is compatible with virtually all UNIX platforms and runs with Windows 95/98/NT and MacOS 8.x.

```
const int Dim = 2; // Dimensionality

// Mesh size; 129 vertices per dimension:
Interval<Dim> vertDomain;
for (int d = 0; d < Dim; d++) vertDomain[d] = Interval<1>(129);

// Uniform, logically-rectilinear mesh; Cartesian coordinates:
typedef UniformRectilinearMesh<Dim, Cartesian<Dim> > Mesh_t;
Vector<Dim> origin(0.0), spacings(0.2);
Mesh_t mesh(vertDomain, origin, spacings);

// Geometry object for cell-centering with respect to mesh,
// one guard layer:
typedef DiscreteGeometry<Cell, Mesh_t > Geometry_t;
GuardLayers<Dim> gl(1);
Geometry_t geom(mesh, gl);

// Parallel layout; 8 patches per dimension:
Loc<Dim> patches(8);
GridLayout<Dim> layout(mesh.physicalCellDomain, patches, gl, gl);

// Create two scalar & one Vector Field (all using multi-patch
// engines with compressible bricks as the patches):
typedef MultiPatch<GridTag, CompressibleBrick> MP_t;
Field<Geometry_t, double, MP_t> s1(geom, layout), s2(geom, layout);
Field<Geometry_t, Vector<Dim>, MP_t> v(geom, layout);

s2 = s2.x()*s2.x(); // Assign s2(x) = x**2

// Assign u1 from s2 and divergence of v:
s1 = s2 + div<Cell>(v);
```

Figure 2. Illustration of some POOMA 2.1 *Field* capabilities. Shows use of *DiscreteGeometry<Mesh,CoordinateSystem>* for the *Field* class's *Geometry* parameter, *Vector* for the *T* parameter, and *MultiPatch* for the *EngineTag* parameter. Also shows use of the high-level divergence operator *div<Cell>(f)*, which returns a *Field* having *Cell* centering with respect to the same mesh as the input field *f*. This inlines with the rest of the expression on the right-hand side of the assignment, via the expression-template system. All of this code is implicitly parallel. It is also dimensionality-independent: the same source is correct for 3D simply by changing the value of Dim from 2 to 3.

```
template <class PTraits>
class Airplanes : public Particles<PTraits> {
  public:
    // typedefs
    typedef typename PTraits::ParticleLayout_t Layout_t;
    typedef typename PTraits::AttributeEngineTag_t EngineTag_t;
    // constructor/destructor
    Airplanes(Layout_t& layout) : Particles<PTraits>(layout) {
     // register each particle attribute with Particles
     addAttribute(Coordinates_m);
     addAttribute(Velocity_m);
     addAttribute(Fuel_m);
    }
    ~Airplanes() { }
    // accessors
    DynamicArray<Vector<3,double>,EngineTag_t>
    coordinates() { return Coordinates_m; }
    DynamicArray<Vector<3,double>,EngineTag_t>
    velocity() { return Velocity_m; }
    DynamicArray<double,EngineTag_t>
    fuel() { return Fuel_m; }
  private:
    // data members
    DynamicArray<Vector<3,double>,EngineTag_t> Coordinates_m;
    DynamicArray<Vector<3,double>,EngineTag_t> Velocity_m;
    DynamicArray<double,EngineTag_t> Fuel_m;
};
```

Figure 3. Illustration of the derivation of a sample *Airplanes* class from the *Particles* base class.

More information about POOMA...

contact: Scott Haney
e-mail: swhaney@lanl.gov
web: www.acl.lanl.gov/pooma/

Get POOMA and other Advanced Computing Laboratory Software...

web: www.acl.lanl.gov/software/
cd: 1999 Advanced Computing Laboratory Software

This work supported by the US Department of Energy.

Los Alamos
NATIONAL LABORATORY

advanced computing laboratory

LALP-99-202 November 1999